MongodB
Course :- 1

PAGE NO.
DATE / /

Introduction to Crud Operations in MongoDB

Objectives :-

- Explain how to perform data modification in Mongo DB
- Explain how to perform batch insert, ordered bulk insert, and unordered bulk insert
- Explain how to insert documents to mongoDB collection
- Explain how to modify, update, upsert and delete documents from MongoDB collection
- Identify the steps to use different query modifiers such as find, Query conditionals, Queries, $ not Regular expressions, and so on
- Identify the steps to retriere documents by using the find query, or condition, cursor, batch insert
- Identify the steps to retriere documents for array fields
- Identify the steps to perform a batch insert, ordered bulk insert, and unordered bulk insert.

* Data modification in MongoDB.

It involve creating, updating or deleting data. These operations modify the data of a single collection. The insert() method is used to insert a document into a MongoDB collection.

Insert query :-

```
db.courses.insert({Name :"simplilearn",
Addres:
courses:
office:      });
```

* Batch insert in MongoDB:

A batch insert allows storing of multiple documents in a database at one time. Following are the characteristics of a batch insert:

- Sends hundreds or thousands of documents to the database in a batch at one time
- Inserts are faster.
- Reduces insert time by eliminating many header processing activity
- Batch inserts are used in applications for storing server logs and sensors data
- Limits insert to 16MB in a single batch insert

* Ordered bulk insert :-

MongoDB groups ordered list operations by its type and contiguity. Characteristics of ordered bulk insert include the following

- Executes the write operations serially
- If an error occurs during one write operation, MongoDB returns the remaining write operations
- Each group of operations can have maximum 1000 operations.
- on exceeding 1000 operations, MongoDB divides the group into smaller groups of 1000 or less.

```
var bulk = db.items.initializeOrderedBulkOp();
        bulk.insert({   });
        bulk.insert({   });
        bulk.insert({   });
        bulk.execute();
```

* Unordered Bulk Insert :-

In an unordered operations list, MongoDB can

execute in parallel in a nondeterministic manner. When performing an unordered list of operations, MongoDB

- Groups and reorderes the operations for enhanced performance
- further split the groups when the number of operations cross 1000 in each group

```
var bulk = db.items.initializeUnorderedBulkOp();
    bulk.insert ({ });
    bulk.insert ({ });
    bulk.insert ({ });
    bulk.execute();
```

* Insert : Internals and Implications

Process of executing an insert operation is as follows :-

- Language drivers converts the data structure into Binary JSON (BSON) before sending to the database
- The database looks for '_id' key and confirms that the document has not exceeded 16MB.
- The document is saved to the database without any changes.

MongoDB does the following before sending data to the database:

- check for invalid data
- checks UTF-8 compliance of strings.
- filter unrecognized datatypes

Lesson 2: Documents Retrieval in MongoDB

* Retrieving the documents:

MongoDB queries define the criteria or conditions for document retrieval. A query may also include a projection that defines the fields that match the document fields to return. You can modify queries to impose limits, skips and sorts orders.

```
db.items.find({available:true},{item:1}).limit(5)
```

* specify Equality condition

The findOne() method finds the first record ina document. The pretty() method helps get properly formatted results. The query given below select all documents in a collection displays the results in proper format.

```
db.items.find().pretty()
```

An empty ({}) query document selects all documents in the collection. The query given below finds all documents in a collection.

```
db.items.find({}) = db.items.find()
```

To specify an equality condition, use the query document {<field>:<value>}. The query below retrieves all documents having the available field values ~~are~~ as true

* $in, $or, and "AND" conditions.

You can specify query conditions using the following query operators:-

- $in: Queries a variety of values for a single key

```
db.items.find({available:{$in:[true,false]}})
```

$or: Queries similar values as $in operator. However, it is recomended to use the $in operator when performing equality checks on the same field

AND:- A compound query can specify conditions for more than one field in the collection's documents.

```
db.items.find({available:true, soldQty:{$lt:900}})
```

$or Operator

following the functions of or operator

- Returns the value true when any of its expressions evaluate to true or accepts any argument expression.
- Defines a compound query where each clause are joined with a logical OR conjunction

```
db.items.find({$or:[{soldQty:{$gt:500}},{available:true}]})
```

specify AND/OR conditions

- select all document in the collection where:

- The value of the available field is true
- The soldQty has a value greater than 200
- The value of the item field is "book"

```
db.items.find({available:true, $or:[{soldQty:{$gt:200}},{item:"Book"}]})
```

The "$not" is a meta conditional operator. You can apply a $not to any other criteria. In the example given below, the "$mod" queries the keys whose values, when divided by the given value shows a remainder of the 2nd value.

```
db.items.find({"_id":{"$not":{"$mod":[4,1]}}})
```

* Regular Expression:

Regular expressions string matches flexible items and performs case insensitive matching

```
db.items.find({item:/pe/i})
```

You can modify the regular expression to search for any variation

```
db.items.find({item:/pen?/i})
```

* Array Exact match:

Equality matches can specify a single element in the array to match. If the array contains minimum one element with the specified value, these specifications match.

The example below queries for documents that contains an array country_codes that contain 5 as one of its elements

```
db.items.find({country_codes:5})
```

In the example below, the query uses the dot notation

```
db.items.find({'country_codes.0':1})
```

* Array Projection operators :-

MongoDB provides three projections operators - $elemMatch, $slice, and $. The operation given below uses the $slice projection operator.

```
db.items.find({_id:5}, {country_codes:{$slice:2}})
```

$elemMatch, $slice, and $ are used to return a subset of elements for an array key. The operation given below uses $elemMatch to get that document in which the country_codes array value is ($gte) and ($lte) to 6.

```
db.items.find({country_codes:{$elemMatch:{$gte:3, $lte:6}}})
```

* $where query

The '$where' clause allows you to do the following
- Represent the queries that key/value pairs fail to represent
- perform any logical execution within a query.
- Compare the values of two keys in a document.

```
>db.foo.insert({"apple":8, "spinach":4, "watermelon":4})
```

* Cursor

Cursor allow you to control a query output by limiting the numbers of results, skipping some results and sorting results. To create a cursor with the shell, add documents, performs a query on the documents and then allocate the results to a local variable, such as "var".

The query is sent to the server.

↓

shell fetches the first 100 or the first 4MB of results.

↓

- shell contacts the databases, requests for more results.

```
var cursor = db.collection.find()
> var cursor = db.people.find();
> cursor.forEach(function(x)
    db.c.find().skip(3)
    db.c.find().sort({username:1, age:-1})
```

* Cursor (contd)

Before sending a query to the database

limit

Set a limit to the number of results fetched.
e.g.: To limit the number of documents found to three, use the query_ db.items.find().limit(3)

skip:-

skip new documents from the results. eg. To skip the first three documents, use the query_ db.c.find().skip(3)

Sort

Sort the results in ascending or descending manner. To sort results by name in the ascending and age in the descending manner, use the query

- db.c.find().sort({username:1, age:-1})

* Pagination:

To display 25 results sorted by price from high to low per page, perform the query given below

db.stock.find({"desc":"mobile"}).limit(25).sort({"price":-1})

To display the next page for more results, use the second query given below:

db.stock.find({"desc":"mobile"}).limit(25.).skip(25).sort({"price":-1})

* Pagination: Avoiding Larger skips

Pagination lets you control the number of documents returned by the find() query. To display the first page of a query result in the descending chronological order, use the query given below

var page1 = db.foo.find().sort({"date":-1}).limit(100)

* Advance query option:

The following options help in adding arbitrary options to queries

$maxscan : Integer:

This specifies the maximum number of documents that are scanned for a query.

$.min : Document

This starts the criteria for querying

$max : Document

This ends the criteria for querying.

* hint: Document

This tells the servers which index to use for the query.

* explain: Boolean

Does not perform the actual query but explains how the query will be executed.

* snapshot Boolean: This forces the query to use the index of the _ID field.

Lesson 3: Update Operations

* Update operations

The db.collection.update() method in MongoDB modifies existing documents in a collection. It identifies the documents that needs update and options that affect its behaviour.

Operations performed by an update are atomic within a single document. This method has the following parameters.

- An update condition
- An update operation
- An options document

Note: You need to use a structure and syntax for specifying the updates condition and use 'multi:true' to update multiple documents.

* $set

MongoDB provides update operators, such as $set to modify values to change a field values You need to perform the following steps to use a $set modifier

1. Use update operators to change field values

```
db.items.update({item:"book"},{$set:{category:
'NOSQL',details:{ISDN:"1234",publisher:"xyz"}}});
```

2. Update an embedded field

```
db.items.update({item:"pen"},{$set:{"details.model":
"14Q2"}})
```

3. Update multiple documents

```
db.items.update({item:"pen"},{$set:{category:
"stationary"},$currentdate:{lastmodified:true}},
{multi:true})
```

* $Unset and $inc modifiers

A $inc operator is used for incrementing and decrementing numbers. The query given below increments the value of the soldQty field by 1 for "pencil".

```
db.items.update({"item":"Pencil"},{"$inc":{"sold
Qty":1}})
```

The value of the "$inc" key must be a number because you cannot increment a non-numeric value To remove the soldQty field for all the documents where it's value is greater than 700, use the query

given below:

```
> db.items.update({soldQty:{$gt:700}},{$unset:{soldQty:
    "1000"}},{multi:true})
```

$Push and $addToset

The $Push function allows you to add new elements into an array field. The $push function does one of the following

- adds the element at the end of an array if the array already exists
- creates an array field and insert the element into that field

The query given below allows you to use the $push function to an ingredients key.

```
db.user.update({"item":"Pencil"},{$push:{"ingredients":
    {"wood":'California cader","graphite":"mixture of
    natural graphite and chemicals'}}})
```

when adding another email address, use the "$addToset" modifier to prevent duplicate.

Lesson 4: Positional Array Modification

* Positional array modification:

You can modify the array values in two ways:

- By position
- By using the position operator ("$")

To increment the votes for the first comment

in a blog post, use the command given below

```
db.blog.update({"post": post_id}, {"$inc": {"comments.0.votes":1}})
```

The positional operator ("$") helps identify the arrays matching the query document and updates an author's name in an existing document for the blog collection.

```
db.blog.update({"comments.author":"John"},
        {"comments.$.author":"Jim"}})
```

Note :- The positional operator will update only the first match in a document

* Adding elements to Array fields using AddToset:

* Upsert

An upsert is a special kind of update that does the following

- updates a document if it matches an update criteria are found.
- Insert new documents into the collection if no matching criteria is found

The command given below is an example of upsert operation.

```
db.items.update({"item":"Bag"},{"$inc":{"soldQty":1}},
                                {upsert:true})
```

To update all the documents that matches the query criteria, you can true as the fourth parameter.

below is an example of a multi update command

```
db.users.update {item : "10/12/1978"}, {$set : {gift :
        "Happy birthday !"}}, false true).
```

* Removing Documents :-

You can remove documents from mongoDB in the following way:

- To remove data from a collection use the command given below

```
db. courses . remove()
```

- To remove documents from the items collection where the value for "item" is "Bag", use the command given below

```
db.items . remove({"item" : "Bag"})
```

- To remove a single document, call the remove() method with the two parameters.

```
db. items . remove({"item" : "Bag"}, 1)
```

Course 2 :- Introduction to indexing and Aggregation in MongoDB

* Indexing :-

objectives :-

- Explain how to create unique, compound, sparse, text and geospatial indexes in mongoDB

- Explain the process of checking the indexes used by MongoDB when retrieving the documents from the database
- Identify the steps to create, remove and modify indexes
- Explain how to manage indexes by listing, modifying and dropping
- Identify different kinds of aggregation tools available in MongoDB
- Explain how to use MapReduce to perform complex aggregation operations in MongoDB.

* Introduction to Indexing

Indexes are data structures that store collection's data set in a form that is easy to traverse. Indexes help perform the following functions

- Execute queries and find documents that match the query criteria without a collection scan
- Limit the number of documents a query examines
- Store field value in the order of the value.
- Support equality matches are range based queries

Note: MongoDB indexes are similar to the indexes in any other databases.

* Types of Index:

MongoDB supports the following index types for querying:

- Default_id:- Each MongoDB collection contains an index on the default_id field.

- single field: for single field index and sort operations MongoDB can traverse the indexes either in the ascending or descending order.

- Compound Index :- MongoDB supports user-defined indexes for multiple fields.

- Multikey Index : Used for indexing array data

- Geospatial Index : Uses 2d indexes and 2d sphere indexes.

- Text Indexes :- Searches data string in a collection.

- Hashed indexes :- MongoDB supports hash based sharding and provides hashed indexes.

* Properties of Index :-

following are the index properties of MongoDB

Unique indexes

Unique indexes ensure that duplicate values for the indexed file are rejected. They can be interchanged functionally with other MongoDB indexes

Sparse indexes

Sparse indexes ensure that queries search document entries having indexed field & documents without indexed fields are skipped

TTL Indexes

Automatically delete documents from a collection after specified duration of time

* single field index:-

MongoDB supports indexes on any document field in a collection. By default, the _id field in all collections have indexes. Moreover, applications and users add indexes for triggering queries and performing operations.

MongoDB supports both, single field or multiple field indexes based on the operations the index-type performs

```
db.items.createIndex({"item":1})
```

* single field Index on Enbedded Document

You can index top level fields within a document. Similarly, you can create indexes within embedded document fields.

```
{"_id":3, "item":"Book", "available":true, "soldQty":
144821, "category":"NoSQL", "details":{"ISDN":"1234",
"publisher":"XYZ company"}, "onlineSale":true}
```

Use the query given below to create an index on the ISDN field and an embedded document.

```
db.items.createIndex({details.ISDN})
```

* Compound Indexes

A compound index in mongoDB contains multiple single field indexes separated by a comma,

```
db.products.createIndex({"Item":1, "stock":1})
```

Note: MongoDB limits the field of a compound index to a maximum of 31.

PAGE No.
DATE / /

* Index Prefixes:-

Index prefixes are created by taking different combination of fields and start from the first field. for example, consider the compound index given below

{"item": 1, "available":1, "soldQty":1}.

MongoDB cannot efficiently support the query on the item and "soldQty" fields by using index prefixes. The item field is a part of the compound index and the index prefixes and therefore should be used in the find query of the index.

* Sort order

The sort operations help retrieve document based on the sort order in an index following are the characteristic

- If sorted documents cannot be obtained from an index, the results will get sorted in the memory.
- Sort operations executed using an index show better performance than those executed without using an index
- Sort operations performed without an index get terminated after exhausting 32MB of memory.
- Indexes store field reference in the ascending or descending sort order
- Sort order is not important for single fields indexes because MongoDB can traverse the index in ~~their~~ either direction.

* Ensure Indexes fit RAM

- For Fast query processing, ensure that your indexes fit into your system RAM.

- To check the index size, use the query given below
  db.collection.totalIndexesSize()

To ensure indexes fit your RAM
- Have more than the required RAM available
- Have ram available for the rest of the working set
- for multiple collections, check the size of all indexes across all collections.

* Multi-key Indexes
  when indexing a field containing an array value MongoDB creates separate index entries for each array component. MongoDB lets you construct multi-key indexes for arrays holding scalar values, such as strings, numbers, and nested documents

  To create a multi-key index, use the method given below
  db.coll.createindex({<field>:<1 or -1>})
  If the indexed field contains an array, MongoDB automatically decides to:

- Create a multi-key index
- Not create a multi-key index

* Compound Multi-key Indexes
  In compound multi-key indexes, each indexed documents can have maximum one indexed field with an array value. When more than one field contain an array value, compound multi-key indexes cannot be created

Given below is an example of document structure

{_id:1, product_id:[1,2], retail_id:[100,200], category: "both fields are arrays"}

Note:- A shard key index and a hashed index cannot be a multikey index.

* Hashed Indexes:-

The hashing function does the following

- Combines all embedded documents
- Computes hashes for all field values.
- Supports sharding, uses a hashed shard key to shard a collection, and ensures an even data distribution.
- Supports equality queries, however, range queries are not supported

You can not create unique or compound index by taking a field whose type is hashed. However, you can create a hashed and non-hashed index for the same field

To create hashed index

```
db.items.createIndex({item:"hashed"})
```

* TTL Indexes

Total Time to Live (TTL) indexes can be created by combining db.collection.createIndex() and "expireAfterSeconds":

To create a TTL index, use the operation.

```
db.eventlog.createIndex({"lastmodifiedDate":1},
                        {expireAfterSeconds:3600})
```

TTL indexes have following limitations.

- Not supported by compound indexes and the _id field does not support TTL indexes.
- Cannot be created on a capped collection.
- Does not allow createIndex() to change the value of "expireAfterseconds" of an existing index

Note: To change a non-TTL single-field index to a TTL index, drop the index and recreate the index with the "expireAfterSeconds" option.

* Unique Indexes:-

Unique indexes can be created by using the db.collection.createIndex() method and set the unique option to true. To create a unique index on the item field of the items collection, execute the operation given below

```
db.items.createIndex({"item":1},{unique:true})
```

If a unique index has no value, the index stores a null value for the document. Because of this unique constraint, mongoDB permits only one document without the indexed field. for more than one document with a valueless or missing indexed field, the index build process fails.

To filter null values in a document and avoid error, combine the unique constraint with the sparse index.

* Sparse Indexes

Sparse indexes manage documents with indexed fields and ignores documents which do not contain any index field.

To create a sparse index, use the db.collection.createIndex() method and set the sparse option to true

```
db.addresses.createIndex({"xmpp_id":1}, {sparse:true})
```

- when a sparse index returns an incomplete index, then MongoDB does not use that index unless it is specified in the hint method.

```
{x:{$exists:false}}
```

Note:- An index combining sparse and unique indexes does not allow duplicate field values for a single field

* Text Indexes

Text indexes in MongoDB helps search for text strings in documents of a collection. To access text indexes, trigger a query using the $text query operator

- when you create text indexes for multiple fields, specify the individual fields or use the wildcart specifier ($**)

To create text indexes on the subject and content fields, perform the query given below

```
db.collection.createIndex({subject:"text", content:"text"})
```

-The wildcard specifier ($**) indexes all fields containing string content. The example given below indexes

any string value available in each field of each document

```
db.collection.createIndex({"$**":"text"}, {name: "textIndex"})
```

* Text search:

MongoDB supports various languages for text search. The text indexes use simple language-specific suffix stemming instead of language-specific stop words, such as "the", "an", "a", "and".

The query given below performs a text search for the iteam field "customer_info" with spanish as the default language

```
db.customer_info.createIndex({"item": "text"},
                {default_language: "spanish"})
```

Note: A compound text index cannot include special index types, such as multi-key or geospatial index fields.

* Index creation:

During index creation, operations on a data base are blocked and the database becomes unavailable for any read or write operation. The ~~end~~ read or write operations on the database queue allow the index building process to complete.

To make MongoDB available even during an index build process, use the command given below.

```
db.items.createIndex({item:1}, {background: true})
```

db.items.createIndex({category:1}, {sparse:true, background:true})

Note: By default, background is false for building MongoDB indexes.

If you perform any administrative operations when MongoDB is creating indexes in the background for a collection, you will receive an error.

The index build process at the background:

- Uses an incremental approach and is slower than the normal "foreground" process.
- depends on the size of the index for its speed.
- Impact database performance

To avoid any performance issues, use:

- getIndexes() to ensure that your application checks for the indexes at the start up.
- Equivalent method for your driver and ensure it terminates an operation if the proper indexes do not exist
- Separate application codes and designated maintenance windows

* Index Creation on Replica Set

Background index operations on a secondary replica set begin after the index build completes in the primary. To build large indexes on secondaries perform the following steps

1) Restart one secondary at a time in a standlone mode

2) The index build completes

3. Restart as a member of replica set

4. Catch up with the other members of the set

5. Build the index on the next secondary

6. When all secondaries have new index, step down the primary

7. Restart as a standalone

8. Build the index on the former primary

Use the command below to specify a name for an index

```
db.products.createIndex ({item:1, quantity:-1}, {name:
                                        "inventory"})
```

* Remove Indexes

Use following method to remove.

dropIndex() method

To remove an ascending index on the item field in the items collection:

```
db.accounts.dropIndex({"tax-id":1})
```

db.collection.dropIndexe() method:

To remove all indexes barring the _id index from a collection, use the command:

```
db.collection.dropIndexes()
```

* Modify Indexes

To modify an index, perform the following steps:

1) Drop index : execute the query given below to return a document showing the operation status

```
db.orders.dropIndex({"cust_id":1,"ord_date":-1,"items":1})
```

2) Recreate the index :-

Execute the query given below to return a document showing the status of the results

```
db.orders.createIndex({"cust_id":1,"ord_date":-1,
                       "items":-1})
```

* Rebuild Indexes :-

To rebuild all indexes of a collection, use the db.collection.reIndex method. This will drop all indexes including _id and rebuild all indexes in a single operation. You can use following commands

- db.currentOp() - Type this command in the mongo shell to view the indexing process status.
- db.killOp() - Type this command in the mongo shell to abort an ongoing index build process

Note :- You cannot abort a replicated index built on the secondary replica set.

* Listing Indexes :-

All indexes of a collection and a database can be listed. To get a list of all indexes of a collection, use the

db.collection.getIndexes() or a similar method

To list all indexes of collections, use the operation given below in the mongo shell.

```
db.getCollectionNames().forEach(function(collection)
```

* Measure Index use

Query performances indicate index usage. MongoDB provides the following methods to observe index use for your database:

- The explain() method – Used to print information about query execution. Returns a document that explain the process and indexes used to return a query.

- db.collection.explain() or the cursor.explain() – helps measure index usages.

* Control Index Use

To force MongoDB to use particular indexes for querying documents, you need to specify the index with the hint() method, which can be appended in the find() method.

The command given below queries a document whose item field value is "Book" and available field is "true".

```
db.items.find({item:"Book", available:true}).hint({item:1})
```

To view the execution statistics for a specific index, use the query below:

```
db.items.find({item:"book", available:true}).hint({item:1}).explain("executionStats")
```

```
db.items.explain("executionStats").find({item:"Book",available
                                    :true}).hint({item:1})
```

To prevent MongoDB from using any index, specify the $natural operator to the hint() method.

```
db.items.find({item:"book",available:true}).hint({$natural:
    1}).explain("executionstats)
```

* Index ~~and~~ Usg reporting :-

MongoDB provides different matrices to report index use and operations.

These metrics are printed using the following commands

• server status

- scanned - Displays the documents that MongoDB scans in the index to carry out the operation
- scan and order : A boolean that is true when a query cannot use the order of documents in the index for returning sorted results

• collStats :

- TotalIndexSize : Returns index size in bytes
- IndexSizes : Explains the size of the data allocated for an index

• dbStats :-

- dbStats.indexes : contains a count of the total number of indexes across all collections in the database.
- dbStats.indexSize :- The total size in bytes of all indexes created on this database

* Geospatial Index

MongoDB provides geospatial indexes for geospatial

queries.

- To find any location from your current location, you need to create a special index and search in two dimensions - longitude and latitude. A geospatial index is created using the createIndex() function. It passes "2d" or "2dsphere" as a value instead of 1 or -1

To query geospatial data, you first need to create geaspatial index. Use the command given below to create a geaspatial index.

db.collection.createindex({<location field>:"2D sphere"})

Note :- A compound index can include a 2dsphere index key in combination with non-geospatial index keys.

* MongoDB's Geospatial Query Operators

The geospatial query operators in mongoDB lets you perform the following queries

1) Inclusion queries
- Return the locations included within a specified polygon
- Use the operator $geoWithin. The 2d and 2dsphere indexes support this query

2) Intersection Queries
- Return locations intersecting with a specified geometry
- Use the $geoIntersects operator and return the data on a spherical surface

* Proximity Queries
- Return various points closer to a specified point.
- Use the $near operator that requires a 2d or 2d sphere index.

* $Geo with operator
The $geo within operator is used to query location data found within a GeoJSON polygon.
Use the syntax given below to use the $geoWithin Operator.

db.<collection>.find({<location field>:
{$geoWithin:{$geometry:{type:"Polygon", coordinates:
[<coordinates>]}}})

The example given below selects all points and shapes that exist entirely within a GeoJSON polygon

db.places.find({loc:{$geoWithin:
{$geometry:{type:"polygon", coordinates:[[[0,0],[3,6],[6,0],[0,0]]}}}})

* Proximity Queries in MongoDB
Proximity queries return the point closest to the specified point and sort the results by its proximity to the specified point
To perform a proximity query on the GeoJSON data points, you need to
- Create a 2dsphere index
- Use the $near or $geonear operator

The $near operator uses the syntax given below
db.<collection>.find({<location field>:{$near:{
$geometry:{type:"point", coordinates:[<longitude>

<latitude>]}.
$maxDistance:<distance in meters>}}})

The geoNear command uses the second syntax given below

db.runcommand({geoNear:<collection>,near:{type:"point",coordinates:[<longitude>,<latitude>]}.
spherical:true})

Lesson 2. Aggregation

* Aggregation:-

Aggregations process data sets and return calculated results. It is run on the mongod instance to simplify applications codes and limit resource requirements. following are the characteristics of aggregation

- Uses collections of documents as an input and return results in the form of one or more documents.

- Is based on data processing pipelines. Document pass through multi-stage pipeline and gets transformed into an aggregated results.

- The most basic pipeline stage in the aggregation frameworks provide filters that function like queries

- The pipeline operations group and sort documents by defined field or fields.

- The pipeline users uses native operations within MongoDB to allow efficient data aggregation and is the favoured method for data aggregation.

* Pipeline operators and Indexes:

The aggregate command in mongoDB function on a single collection and logically passes the collection through the aggregation pipeline. Use the $match, $limit and $skip stages to optimize aggregate operations.

You may require only a subset of data from a collection to perform an aggregation operation. Therefore use the $match, $limit and $skip stages to filter the documents. The $match operation scans and selects only the matching documents in a collection when placed at the beginning of a pipeline.

Placing a $match pipeline stages followed by a $sort stage at the beginning of the pipeline is equivalent to a single query with a sort and can use an index. Place $match operators at the beginning of the pipeline if possible

* Aggregate pipe line stages

Documents are passed through the pipeline stages in a proper order one after the other. The various pipeline stages are:

$project:-

Adds new fields or remove existing fields and thus restructure each document in the stream. It returns one output document for each input

document provided

$match

Filters the document stream and allows only matching documents to pass into the next stage. for each input document, it returns one output document if there is a match or zero if no match is found.

$group

Groups document based on the specified identifier expression and applies logic known as accumulator expression to compute the output document

$sort:

Rearranges the order of the document stream using specified sort keys. Provides one output document for each input document.

$skip:

skips n number of documents and passes the remaining documents without any modifications to the pipeline. Returns either zero documents for the first n documents or one document or one document

$limit

Passes the first n number of documents without any modifications to the pipeline. Return the either one document for the first n documents or zero documents after the first n documents

$unwind

Deconstruct an array field in the input document.

to return a document for each document.

* Aggregation Example

The aggregation operation given below returns all stats with total population greater than 10 million

```
db.zipcode.aggregate([{$group:{_id:"$state", totalPop:{$sum:
"$pop"}}},
{$match:{totalPop:{$gte:10*1000*1000}}}])
```

In this operation, the $group stage does the following:

- Groups the documents of the zipcode collection by the state field
- Calculates "thetotalPop" field for each state
- Returns an output document for each unique state

The aggregation operation given below returns user names sorted by the month of their joining

```
db.users.aggregate([{$project:{month_joined:{$month:
"$joined"}},name:"$_id",_id:0}},{$sort:{month_joined
:1}}])
```

* MapReduce

MapReduce is a data processing model used for aggregation. To perform map-reduce operations, MongoDB provides the MapReduce database command.

A MapReduce operation consist of the following two phases:

- Map stage: Documents are processed and one or more object are produced for each input

document.

- Reduce stage:-

The outputs of the map operation are combined.

Optionally, there can be additional stage to make final modification to the results.

Note: MapReduce can define a query condition to select the input documents, and sort and limit the results.

The MapReduce function in MongoDB can be written as Javascript codes. Typically, MapReduce operations.

- Accept documents as input, performs sort and limit function, and then start the map stage. At the end of a MapReduce operation the result is generated as documents which can be saved in the collection.

- Associate values to a key by using the custom Java script functions. If a key contains more than one mapped value, then this operation converts them into a single object, such as an array.

The use of custom Javascript functions make the MapReduce operation flexible. the MapReduce operations use the custom Javascript function to alter the results at the conclusion of the map and reduce operations may perform further calculations.

* Aggregation Operations:-

Aggregation operations manipulate data and return a compound result based on the input document and

a specific procedure. It provides the following semantics for data processing:

Count

This command along with the two methods, count() and cursor.count() provide access to total counts in the mongo shell. The command given below counts all documents in the "customers_info" collection

```
db.customers_info.count()
```

distinct:

This operation searches for documents matching a query and returns all unique values for a field in the matched document. The syntax given below is an example of a distinct operation.

```
db.customers_info.distinct("customer_name")
```

-Group operation accept sets of documents as input which matches the given query, apply the operation and then return array of documents with the computed results.

A group does not support sharded collection data. In addition, the results of the group operation must not exceed 16 megabytes.

The group operation shown below groups documents by the field 'a', where 'a' is less than three and sums the field count for each group

```
db.records.group({key:{a:1}, cond:{a:{$lt:3}},
reduce: function(cur, results){result.count += cur.count},
initial:{count:0}})
```